

# Night Owl

Oliver Katz

March 5, 2013

## 1 Design

Night Owl is a virtual machine which uses just-in-time (JIT) compilation to generate native machine code at runtime. Night Owl's kernel can be accessed through assembly easily, allowing Night Owl programs to generate other Night Owl programs within themselves with a low overhead.

The kernel contains little more than the code necessary for emitting machine code instructions into a memory block. Using the C++ library (`build.h`) provides utilities for generating code easier as well as optimizations for that code. This is done in a single pass (optimizations included).

Night Owl is designed to be flexible, in that it can run at many levels of abstraction and overhead; it can be as minimal as you want.

## 2 Using the Kernel

Night Owl is a register machine, which means that you need to use registers in your generated code. You can do this automatically with the `Builder` class, but not in the kernel. To use the kernel, you need to create an `o_jit` structure, which contains the generated machine code.

```
o_jit jit = o_jit_init();
```

There are a group of functions called emitters. They all have the prototype `void emitter(o_jit *jit, unsigned char r0, unsigned char r1, int a)`, where `r0` is the first register parameter, `r1` is the second, and `a` is the integer parameter. If the emitter doesn't use a particular parameter, then it will still require you to provide it, but its value will not be taken into account.

### 2.1 Emitters

The first emitter is for the NOP instruction. Simply put, it does nothing. It simply skips to the next CPU cycle. You can emit this instruction with:

```
o_emit_nop(&jit, 0, 0, 0);
```

The following is a reference of all of the instructions:

Emitter	Assembly Instructions	Description
<code>o_emit_nop</code>	???	Skips a CPU cycle and does nothing
<code>o_emit_const</code>	<code>mov, movi</code>	Store a into r0
<code>o_emit_mov</code>	<code>mov</code>	Move the value of r1 into r0
<code>o_emit_mov_disp</code>	<code>mov, movd</code>	Dereference the address in (r1 + a)
<code>o_emit_ld</code>	<code>mov, ld</code>	Dereference the address in r1 and store into r0
<code>o_emit_st</code>	<code>mov, st</code>	Store the value in r1 to the address in r0
<code>o_emit_add</code>	<code>add</code>	Add r0 to r1 and store in r0
<code>o_emit_sub</code>	<code>sub</code>	Subtract r1 from r0 and store in r0
<code>o_emit_mul</code>	<code>mul</code>	Multiply r0 by r1 and store in r0
<code>o_emit_and</code>	<code>and</code>	Logical-AND r0 and r1 and store in r0
<code>o_emit_or</code>	<code>or</code>	Logical-OR r0 and r1 and store in r0
<code>o_emit_xor</code>	<code>xor</code>	Logical-EXCLUSIVE-OR r0 and r1 and store in r0
<code>o_emit_pusharg</code>	<code>push, pusha, pusharg</code>	Push the value of r0 to the argument stack
<code>o_emit_poparg</code>	<code>pop, popa, poparg</code>	Pop the top value of the argument stack into r0
<code>o_emit_leave</code>	<code>leave</code>	Leave a function (must be put before the return instruction)
<code>o_emit_ret</code>	<code>ret</code>	Return the value of the EAX register (register 0) from the current function
<code>o_emit_int</code>	<code>int</code>	Call a system interrupt (relies on the values of EAX and EBX)
<code>o_emit_cmp</code>	<code>cmp</code>	Compare r0 and a for use in the branch instructions (like <code>breq</code> )
<code>o_emit_breq</code>	<code>je</code>	If the last values compared are equal, jump to the code position a
<code>o_emit_brne</code>	<code>jne</code>	If the last values compared are unequal, jump to the code position a
<code>o_emit_brlt</code>	<code>jl</code>	If the last register compared is less than the integer it is compared to, jump to the code position a
<code>o_emit_brle</code>	<code>jle</code>	If the last register compared is less than or equal to the integer it is compared to, jump to the code position a
<code>o_emit_brgt</code>	<code>jg</code>	If the last register compared is greater than the integer it is compared to, jump to the code position a
<code>o_emit_brge</code>	<code>jge</code>	If the last register compared is greater than or equal to the integer it is compared to, jump to the code position a
<code>o_emit_jump</code>	<code>jmp, jump</code>	Jump to the code position a
<code>o_emit_call</code>	<code>call</code>	Call the function pointer a
<code>o_emit_fld</code>	<code>fld</code>	Push the double at address r0 to the floating-point stack
<code>o_emit_fst</code>	<code>fst</code>	Pop the double at the top of the floating-point stack to the address r0
<code>o_emit_fadd</code>	<code>fadd</code>	Add the top two doubles on the floating-point stack
<code>o_emit_fsub</code>	<code>fsub</code>	Subtract the top two doubles on the floating-point stack
<code>o_emit_fmul</code>	<code>fmul</code>	Multiply the top two doubles on the floating-point stack
<code>o_emit_fdiv</code>	<code>fdiv</code>	Divide the top two doubles on the floating-point stack
<code>o_emit_fcmp</code>	???	Compare the top two doubles on the floating-point stack (standard branch instructions still work with this)
<code>o_emit_fchs</code>	<code>fchs</code>	Change the sign of the top double on the floating-point stack
<code>o_emit_fst_rot</code>	<code>fdecfpst</code>	Pop the top element of the floating-point stack and push it to the bottom
<code>o_emit_fnop</code>	???	Floating point operations run on an alternate sub-processor called the FPU - this instruction is skips a cycle on that sub-processor and does nothing
<code>o_emit_ftan</code>	???	Take the tangent of the top double on the floating-point stack and push it to the top
<code>o_emit_fatan</code>	???	Take the arc-tangent of the top double on the floating-point stack and push it to the top
<code>o_emit_fsin</code>	???	Take the sine of the top double on the floating-point stack and push it to the top
<code>o_emit_fcos</code>	???	Take the cosine of the top double on the floating-point stack and push it to the top
<code>o_emit_fsincos</code>	???	Take the sine and cosine of the top double on the floating-point stack and push both to the top (faster than calling both instructions individually)
<code>o_emit_fsqrt</code>	???	Take the square root of the top double on the floating-point stack and push it to the top

## 2.2 Registers

There are eight registers that you can use: EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI. Most of these serve special purposes some of the time, while some of them are general purpose. EAX, ECX, EDX, and EBX are generally quite general purpose, except EAX stores the information generated by comparison instructions (`cmp` and `fcmp`), `ret`

returns the value stored in EAX, and `int` uses EAX and EBX. ESP points to the top of the stack at all times. EBP is the base register for functions and argument handling. ESI and EDI are not used within Night Owl, but cannot be used for general purpose operations.

## 2.3 Examples

If you want to make a C-compatible function that returns a sum of its two integer arguments, you can do it as follows:

```
o_jit jit = o_jit_init();
o_emit_pusharg(&jit, 0_EBP, 0, 0);
o_emit_mov(&jit, 0_EBP, 0_ESP, 0);
o_emit_mov_disp(&jit, 0_EAX, 0_EBP, 8);
o_emit_mov_disp(&jit, 0_EBX, 0_EBP, 12);
o_emit_add(&jit, 0_EAX, 0_EBX, 0);
o_emit_leave(&jit, 0, 0, 0);
o_emit_ret(&jit, 0, 0, 0);
```

If you want to make an if-statement, you can do that as follows:

```
// if 5 == 5,
o_emit_const(&jit, 0_EBX, 0, 5);
o_emit_cmp(&jit, 0_EBX, 0, 5);
o_emit_brne(&jit, 0, 0, 5); // branch to else-segment

// then-segment:
o_emit_const(&jit, 0_ECX, 0, 1);
o_emit_jump(&jit, 0, 0, 6);

// else-segment: (code position: [5])
o_emit_const(&jit, 0_ECX, 0, 0);

// end of if-statement (code position: [6])
// ecx should contain 1, because the if statement went into its then-segment
```

After we are done emitting our code, we need to run it. We can do this using a function pointer to the emitted code block in memory:

```
void *func = o_jit_fp(&jit);
```

Or if we have defined multiple functions, you need to add the code position of the beginning of the function to the function pointer. To get the code position at the head of the generation, use `o_jit_label(&jit);`.

## 3 The Builder Class

The Builder class is defined within `builder.h`. It is written in C++ and contains a higher overhead than the core kernel, but it can automate some common tasks and provide optimizations. The most important optimizations it provides are function inlining and lazily-evaluated register and floating-point operations.

Here is an example of how lazily-evaluated operations work. If you have two registers, EAX and EBX and you add their contents together, you get a result which takes three CPU cycles to calculate. If you know the values of EAX and EBX within the program (they are constant), you don't need to run the calculation at run time, you can run it at compile time by finding their sum and replacing the three instructions with one: store the integral result of the sum in EAX as a constant.

Let us create a Builder class:

```
Builder b = Builder();
```

We can emit instructions directly into the builder:

```
b.emit(_add, 0_EAX, 0_EBX, 0);
```

We can also store them temporarily in BuilderInstructions (this has the same result):

```
BuilderInstruction inst = BuilderInstruction(_add, 0_EAX, 0_EBX, 0);  
b.emit(inst);
```

And lastly, we can store many instructions as BuilderInstructionPages:

```
BuilderInstructionPage page = BuilderInstructionPage();  
page << BuilderInstruction(_add, 0_EAX, 0_EBX, 0);  
b.emit(page);
```

If we store multiple functions in a BuilderInstructionPage, as long as we use `function("theFunctionName");` at the beginning of each function, we can get the function segment with `page.getFunctionSegment("theFunctionName");`.

We can use Builders to generate machine code using the kernel, or we can output the generated instructions to a BuilderInstructionPage with:

```
b.outputToPage();
```

### 3.1 Function Inlining

As long as this is enabled and we have only defined a single function within the builder, we can convert the contents of the builder to an instruction page as an inline function call. You can specify the arguments as integer references:

```
int arg0 = 5;  
int arg1 = 10;  
  
vector<int *> tmp;  
tmp.push_back(&arg0);  
tmp.push_back(&arg1);  
  
b.convertToInline(tmp);
```

### 3.2 Function-Generation Utilities

Instead of generating all of the instructions for generating function argument handling in a C-compatible way (as shown above in the kernel examples in section 2.3), we can use the following:

```
b.emitFunction();  
unsigned char r0 = b.emitFunctionArgument(4); // 4 is the size of the argument in bytes (this is an int)  
unsigned char r1 = b.emitFunctionArgument(4);  
b.emit(_add, r0, r1, 0);  
b.emit(_mov, 0_EAX, r0, 0); // r0 is probably eax, but to make sure (if it is, this instruction won't be g  
b.emitFunctionReturn();
```

### 3.3 Relocation Utilities

Jumps and branches often jump to code positions that aren't generated yet. We can use the relocation utilities to handle this automatically.

```
b.emit(_const, 0_EBX, 0, 5);
b.emit(_cmp, 0_EBX, 0, 5);

b.emitRelocatableBranch("ifStatement", "elseSegment");
b.emit(_const, 0_ECX, 0, 1);
b.emitRelocatableJump("elseJump", "endIfStatement");

b.emitRelocatableLabel("elseSegment");
b.emit(_const, 0_ECX, 0, 0);

b.emitRelocatableLabel("endIfStatement");
```

### 3.4 Lazily-Evaluated Operations

Let us start with the difference between dirty and pure registers. A pure register is one to which we know the value at compile-time. A dirty one is any other register. Here is an example of a pure register with the value of 5:

```
unsigned char r = b.regAlloc(false);
b.regValue(r, 5);
```

This register IS NOT emitted into the machine code yet. To do that, we need to call `b.emitReg(r)`; This will make `r` dirty. We can operate on pure registers with:

```
b.emitOperation(_add, r0, r1); // assuming we have registers r0 and r1
```

`emitOperation` works on both pure and dirty registers. We can push pure or dirty registers to the function argument stack with:

```
b.emitPushArg(r0);
```

We can do the same thing with floating-point stack elements.

```
b.fpDirty(0, false); // make the top of the floating-point stack pure
b.fpValue(0, 5.0); // set the value
b.emitFp();
b.emitFloatingOperation(_fchs);
```

You will notice that there is no example of adding new floating-point stack elements. You can do this with the constant and reference utilities.

### 3.5 Constant and Reference Utilities

We can add constant integers (32-bits) with:

```
unsigned char r = b.emitConstI32(5);
```

Or a reference to an integer with:

```
unsigned char r = b.emitConstI32(&anInteger);
```

We can pull the value back to a reference with:

```
b.emitPullI32(r, &anInteger);
```

We can do the same things with the floating-point stack:

```
b.emitRefF64(&aDouble);  
b.emitPullF64();
```

### 3.6 Error Handling

If, while emitting machine code, errors occur, you can find them with the following functions:

```
while(b.countErrors() > 0)  
Builder::error e = b.popError();
```

## 4 Conclusion

I hope this tutorial has helped you learn how to use Night Owl. Please look at the documentation for more information on use or email me any questions that you may have at: [olivetti.katz@gmail.com](mailto:olivetti.katz@gmail.com). Thank you for reading.